Universal Boat Builder Framework
System Outline and Roadmap

Reference guideline for building a reusable, multi-client boat configurator (Prodigy, Phoenix, and future brands)

Version: v0.3   Date: 2026-02-12   Owner: Launch Pad Creative Group

Change log: v0.3 adds Appendix E (compile step specification and publish pipeline).

1. Executive summary

We will build a standalone, multi-tenant web application that functions like a highly-configurable form with advanced conditional logic, real-time pricing, and layered image previews. Each boat brand (client) and model will be defined by versioned configuration data (not custom code), allowing the same Builder Engine to be reused across Prodigy, Phoenix, and additional clients.

Key decisions:

- Architecture: Config-driven Builder Engine + Admin Dashboard + Outputs (PDF/images).
- Stack: Next.js (React) + TypeScript, Payload CMS (admin/auth/media), PostgreSQL, S3-compatible object storage.
- Hybrid imagery: client-side live stacking for instant feedback; server-side compositing (Sharp) for final images/PDFs.
- Rules: deterministic JSON rules evaluated against normalized state with stable IDs (never labels).
- Pricing: line-item engine with packages, conditional surcharges, and snapshot locking at submission time.
- Versioning: all submissions reference a specific ModelVersion so historical quotes/build sheets remain accurate.

This document defines the system contracts, the major components, and a build roadmap that prioritizes a solid foundation before the first Prodigy logic tree is entered.

2. Goals and non-goals

2.1 Goals

- Reusable framework: onboard new clients/models primarily through configuration and assets.
- Clean separation of concerns: schema (form), rules (logic), pricing, and rendering.
- Fast user experience: instant visual feedback and pricing updates as selections change.

- Reliable outputs: generate consistent customer quote PDFs and internal build sheets.
- Maintainability: versioning, auditability, and testability for complex logic trees.

2.2 Non-goals (initially)

- Full visual rule-builder UI in v0 (start with validated JSON rules + test harness).
- Real-time dealer inventory, ERP integration, or production scheduling integrations.
- Perfectly styled marketing PDFs in v0 (start clean and correct; iterate on design later).
- Server-side image compositing on every click (hybrid approach avoids this).

3. Core requirements

3.1 Customer experience

- Multi-step builder flow (steps like hull colors, interior, electronics, engine, etc.), with ability to add model-specific steps/sections.
- Conditional visibility and availability (show/hide fields, enable/disable options, require fields, mutual exclusions).
- Real-time price updates with a transparent line-item breakdown and running total.
- Live layered previews for profile and overhead views (stacked transparent PNG/WebP).
- Optional detail previews (popup galleries) for selections that are hard to see on full-boat renders.
- Submission confirmation and downloadable quote PDF.

3.2 Internal (builder/dealer) experience

- Admin dashboard to create clients, models, steps, fields, options, logic rules, pricing, and layers.
- Media management for layer assets and gallery assets.
- Versioning workflow: draft/edit -> publish -> submissions locked to the published version at time of submission.
- Generate internal build sheet PDF (order form/build sheet) suitable for production use.
- Export capabilities (JSON/CSV) for backups and bulk edits.

4. Architecture overview

The system is divided into three layers: Config, Engine, Outputs.

4.1 Config layer (data)

- Client: branding, contact, disclaimers, theme settings.
- Model and ModelVersion: steps/sections/fields/options, rules, pricing definitions, layers, galleries.
- Assets: PNG/WebP for layers, plus optional gallery media.

4.2 Engine layer (shared logic)

- Form renderer: renders UI from ModelConfig (no hard-coded model logic).

- Rules engine: evaluates conditions and produces computed UI state (visibility, availability, requirements) and state corrections.
- Pricing engine: computes line items and totals from current state.
- Rendering engine: selects active image layers per view; triggers detail gallery previews when configured.

4.3 Outputs layer (documents + images)

- Submissions stored as immutable snapshots (state + computed pricing) tied to a ModelVersion.
- Server-side image compositing (Sharp) to create final hero images for PDF and archival.
- Two PDF templates: customer quote and internal build sheet.

5. Technology stack

Chosen stack aligned with the team's JavaScript/TypeScript and frontend strengths:

Area	Choice
Frontend/UI	Next.js (React) + TypeScript
Admin dashboard	Payload CMS (auth, admin UI, media, access control)
Database	PostgreSQL (JSONB for flexible config + relational for queries)
Object storage	S3-compatible (S3 / Cloudflare R2 / Wasabi)
Server image compositing	Sharp (Node)
PDF generation	HTML templates rendered to PDF (Playwright or Puppeteer)
Shared business logic	packages/engine (rules + pricing + normalization + tests)

6. Core concepts and contracts

6.1 Terminology

Client: A boat brand (e.g., Prodigy, Phoenix) with its own models, branding, and policies.

Model: A boat model (e.g., 21' X-series) that has one or more versioned configurations.

ModelVersion: A versioned snapshot of the model configuration (form schema + rules + pricing + layers).

Builder state: The user's current selections in normalized form (IDs, not labels).

Computed UI state: Derived UI behavior from rules (hidden fields, disabled options, required fields, etc.).

Line items: Pricing breakdown entries computed from base price, options, packages, and conditional surcharges.

Views: Named visual perspectives (profile, overhead, optional detail views).

Layer: A transparent asset drawn in a given order when its conditions match the state.

Gallery: A popup preview for a selection that is hard to see in the hero views.

6.2 ModelConfig contract (compiled config)

Although Payload stores configuration in collections, the Builder UI consumes a compiled ModelConfig object. This compiled config is the stable contract between Admin and Engine.

ModelConfig includes:

- Metadata: clientId, modelId, modelVersionId, name, images sizing, disclaimers.
- Steps/sections/fields: order, labels, help text, validation, field types.
- Options: stable ids/codes, labels, pricing fields, metadata (tags, swatches).
- Rules: prioritized, deterministic JSON rules (when/then).
- Mappings/derived fields: compatibility matrices and computed values.
- Rendering: views, layer groups, layers (z-index + conditions), galleries.
- Pricing configuration: base price and rule-driven line items.

7. Form schema design

Each ModelVersion defines a 3-level structure: Steps -> Sections -> Fields. This supports both standard flows and model-specific custom sections without code changes.

7.1 Field types (initial set)

- single_select (radio/card select)
- multi_select (checkbox group)
- toggle (boolean on/off)
- number (quantity, length, etc.)
- text (rare; for notes where allowed)
- computed (read-only derived values shown to user)

7.2 Stable identifiers

All fields and options must have stable identifiers (UUID or slug). Rules, pricing, and rendering conditions must reference IDs/codes, never display labels.

8. Rules engine

8.1 Purpose

The rules engine turns a user's selections into a computed UI state: what is visible, available, required, and what should be auto-set or cleared to maintain a valid configuration.

8.2 Rule format

Rules are stored as JSON with a 'when' condition and 'then' actions.

```
{
 "id": "rule-hardtop-frame-color-visible",
 "priority": 200,
 "when": { "eq": ["field.hardtop", true] },
 "then": [
  { "action": "showField", "fieldId": "hardtop_frame_color" }
 ]
}
```

8.3 Condition operators (v0)

- all
- any
- not
- eq
- neq
- in
- contains
- gt
- lt

8.4 Actions (v0)

- showStep / hideStep
- showSection / hideSection
- showField / hideField
- enableOption / disableOption
- requireField / unrequireField
- setValue (auto-select) / clearValue (auto-clear)
- addLineItem / removeLineItem (pricing hooks)

8.5 Evaluation order and stability

- Input state is normalized (IDs, arrays sorted, nulls consistent).
- Rules are applied in priority order to produce computed UI state.
- Invalid selections are cleared if they become hidden/disabled.
- Auto-set actions (packages, forced selections) are applied.
- Re-run rules until state reaches a stable fixpoint (cap iterations, e.g., 10) to avoid infinite loops.
- Final output is: normalized state + computed UI state + validation results.

8.6 Priority and overrides

Rules support layered overrides using priority and scope. Higher priority rules win when conflicts occur. Recommended precedence: Engine defaults < Client < Model < Version < Emergency override.

8.7 Debuggability

- Each rule and action must be traceable (rule id, why it fired).
- Provide a 'simulate' tool in admin (later milestone) that shows state + which rules fired.
- Add unit tests for rule evaluation using known sample configurations.

9. Derived values and compatibility mappings

Some behaviors are best modeled as derived values or compatibility matrices rather than many one-off rules. Examples: deck covering color must match upholstery; engine determines allowed rigging packages.

9.1 Derived fields

Use derived/computed fields when a value must always follow another value.

```
// Example: deck covering color derived from upholstery color
derivedFields: [
  {
    "fieldId": "deck_covering_color",
    "derive": { "fromFieldId": "upholstery_color", "map": "deckFromUpholstery" }
  }
]
```

9.2 Compatibility mappings

Use mappings to constrain allowed options and simplify rules.

```
// Example: mapping table (stored as data)
mappings: {
  "deckFromUpholstery": {
    "sand": "mocha",
    "white": "gray",
    "black": "black"
  }
}
```

10. Pricing engine

Pricing is computed as a line-item breakdown rather than a simple sum of selected options. This supports packages, conditional surcharges, and clear build-sheet outputs.

10.1 Pricing inputs

- Base model price (may vary by trim).
- Option prices (MSRP, dealer, or both depending on use case).
- Package definitions (a package may include options and/or add its own price).

- Conditional line items (e.g., if engine = X add rigging surcharge).
- Overrides (rare): explicit price override for a configuration when required by business rules.

10.2 Pricing outputs

- Computed line items with codes, labels, quantities, amounts.
- Subtotal and total (optional taxes/fees later).
- Pricing snapshot stored with the submission (immutable).

11. Rendering system (images + galleries)

Rendering is driven by layers and galleries, not by directly attaching a single image to each option. Many options may have no visual impact; others may affect one or more layers; some may trigger detail previews.

11.1 Views

- profile
- overhead
- optional: detail/closeup views

11.2 Layers

Each layer has: view, asset, z-index (order), and a condition expression.

```
{
 "id": "layer-hardtop-frame-black-profile",
 "view": "profile",
 "assetId": "media/hardtop/frame_black.webp",
 "z": 420,
 "when": { "all": [
  { "eq": ["field.hardtop", true] },
  { "eq": ["field.hardtop_frame_color", "black"] }
 ]}
}
```

11.3 Layer groups

Use layer groups to keep admin manageable (e.g., a 'Hard Top' group containing base + frame variants). Groups are optional but recommended for complex assemblies.

11.4 Galleries (detail previews)

Selections that are subtle in full-boat views can open a modal gallery. Galleries can be attached to options without affecting layers.

- Use cases: upholstery texture, stitching, small hardware, electronics screens, flooring texture.
- Config includes: media items, captions, and trigger rules (e.g., open gallery from an info icon).

11.5 Hybrid approach

- Client-side: stack transparent images instantly for interactive preview (no server round-trips).
- Server-side: composite final images using Sharp on submit / generate-PDF only.
- Cache composed results by a render key (hash of modelVersionId + view + normalized state + asset version).

12. Data storage and submissions

12.1 Config storage

- Payload stores client/model/version entities and media metadata.
- A compile step produces ModelConfig for the Builder UI (per ModelVersion).
- Published versions are immutable; edits create a new draft version.

12.2 Submissions

- A submission stores: clientId, modelId, modelVersionId, timestamp, and user contact info (if captured).
- Store an immutable state snapshot and computed line items (locked pricing).
- Store output artifact references (PDF URLs, composed image URLs).
- Optional: draft/save-and-resume workflow (later milestone).

13. Admin dashboard (Payload)

The admin dashboard replaces ACF/Fluent usage with a dedicated model builder UI. In v0, complex rule authoring can be JSON-based with validation; later iterations can add a visual builder.

13.1 Core collections

- Clients
- Models
- ModelVersions (draft/published)
- Assets (media library)
- Submissions

13.2 Essential admin workflows

- Create client -> create model -> create ModelVersion (draft).
- Define steps/sections/fields/options, assign pricing and codes.
- Upload assets and assign layers per view with conditions.
- Enter rules and mappings; validate and simulate configuration behavior.
- Publish ModelVersion; submissions now reference this version.

13.3 Bulk operations

- Import/export JSON for model versions.

- Optional CSV import for options/pricing to speed onboarding.
- Clone ModelVersion to create next season/next pricing update quickly.

14. Multi-tenant and branding

- Clients have branding settings: logo, colors, typography, disclaimers, dealer contact info.
- The Builder UI uses theme tokens per client (no per-client forks).
- Optional per-client domains/subdomains can be added later (routing by client slug).

15. Testing and quality strategy

- Unit tests for rules evaluation and pricing calculation (packages/engine).
- Golden test cases per model (known states -> expected visible fields, disabled options, totals).
- E2E tests for the builder flow (smoke tests for critical paths).
- PDF output verification (template regression checks).

16. Build roadmap (milestones)

The roadmap is organized as sequential milestones. Each milestone has clear deliverables and acceptance criteria.

Milestone	Deliverables	Acceptance criteria
M0: Foundation	• Repo + Next.js + TypeScript + linting/formatting • Postgres connected; basic environment setup • Payload running (auth + admin UI + media upload) • packages/engine scaffolded with test runner	You can log into admin, create basic entities, and run the app locally.
M1: Config contract + versioning	• Model/ModelVersion schema • Compile ModelVersion -> ModelConfig API endpoint • Draft/published workflow (one published per model) • Submission references ModelVersion	You can publish a model version and fetch a compiled config from the frontend.
M2: Builder UI v0	• Wizard rendering Steps/Sections/Fields from ModelConfig • Initial field types implemented • Validation framework (required fields)	A model can be configured end-to-end without conditional rules.
M3: Rules engine v0 + mapping support	• Condition operators and core actions implemented • Stability loop and auto-clear invalid selections	Conditional logic works; invalid combinations are prevented; tests pass.

	• Derived fields + compatibility mapping mechanism • Rule unit tests + sample fixtures	
M4: Pricing + line items	• Base price + option pricing + packages • Conditional surcharges via rules/line items • Real-time totals + breakdown UI • Pricing snapshot stored with submission	Total price is correct and explainable; submissions lock pricing.
M5: Rendering + outputs	• Client-side stacked previews (profile + overhead) • Layer conditions + layer groups • Gallery modal support • Server composite endpoint (Sharp) for final images • PDF generation (quote + build sheet)	Users see previews, can submit, and receive PDFs; internal build sheet is generated.
M6: Admin usability improvements	• Rule validation + simulation screen • Bulk import/export tools • Better layer management UI (grouping, previews) • Audit logs for config changes	Admin can manage complex models efficiently without developer intervention.

17. Open questions / inputs needed from Prodigy

- Logic tree format: can Prodigy provide it as a spreadsheet, diagram, or structured list of conditions and outcomes?
- Option codes/SKUs: are there standardized codes used on build sheets today?
- Pricing rules: are there package discounts, conditional surcharges, or dealer-specific pricing requirements?
- Asset list: required views (profile/overhead only or additional angles), image sizes, naming conventions, and layer ordering.
- Submission flow: collect dealer assignment, customer contact, deposits, or just quote requests?

Appendix A: Suggested naming conventions

Consistent naming avoids confusion across rules, pricing, and rendering.

- field ids: snake_case (e.g., upholstery_color, deck_covering, hardtop_frame_color)
- option ids: stable slug or UUID; include code where helpful (e.g., black_powdercoat)
- assets: view/group/variant (e.g., profile/hardtop/frame_black.webp)
- rule ids: rule-<topic>-<intent> (e.g., rule-engine-requires-rigging)

Appendix B: Example end-to-end flow (hard top + frame color)

Example: hard top toggle controls visibility and rendering; frame color drives a variant layer.

```
Fields:
- hardtop (toggle)
- hardtop_frame_color (single_select; options: silver, white, black)

Rules:
- If hardtop = false: hide hardtop_frame_color and clear its value
- If hardtop = true: show hardtop_frame_color and require it

Layers (profile):
- hardtop_base when hardtop = true
- hardtop_frame_<color> when hardtop = true AND hardtop_frame_color = <color>
```

Appendix C: ModelConfig TypeScript interface (compiled contract)

This appendix defines the canonical TypeScript contract that the Builder UI and Engine consume. Payload (or any admin system) may store config across multiple collections/tables, but every published ModelVersion must compile into a single ModelConfig object that matches this interface. This contract is the foundation that allows reuse across clients.

C.1 Design principles

- Stable identifiers: rules, pricing, and rendering reference ids/codes, never display labels.
- Compiled config: the UI should not need to query multiple endpoints to render a model; load one ModelConfig for the active ModelVersion.
- Version pinning: submissions always reference modelVersionId and store a state+pricing snapshot for historical accuracy.
- Separation of concerns: form schema (steps/fields/options) is separate from rules, pricing, and rendering definitions.

C.2 TypeScript contract (ModelConfig)

Copy/paste reference for the shared Engine package. This is the interface the Builder UI renders from.

```
// packages/engine/src/model-config.ts

export type ViewId = 'profile' | 'overhead' | 'detail';
export type FieldType =
  | 'single_select'
  | 'multi_select'
  | 'toggle'
  | 'number'
  | 'text'
  | 'package_select'
  | 'computed';

export type PriceMode = 'msrp' | 'dealer' | 'both';

export interface ModelConfig {
  meta: {
    clientId: string;
    clientSlug: string;
    modelId: string;
    modelSlug: string;
    modelVersionId: string;
    versionLabel: string; // e.g., '2026 MSRP v1'
    publishedAtISO: string; // ISO string
    locale?: string; // default 'en-US'
    currency?: string; // default 'USD'
```

```
};

branding: {
  logoUrl?: string;
  primaryColorHex?: string;
  disclaimerText?: string;
  contactEmail?: string;
  contactPhone?: string;
};

form: {
  steps: StepDef[];
  // Optional lookup for UI convenience. Engine should not require this.
  fieldsById?: Record<string, FieldDef>;
};

rules: {
  // A compiled list of rules already merged across scopes (client/model/version).
  // priority resolves conflicts: higher priority wins.
  rules: RuleDef[];
};

derived?: {
  derivedFields?: DerivedFieldDef[];
  mappings?: Record<string, Record<string, unknown>>; // compatibility/mapping tables
};

pricing: {
  mode: PriceMode;
  basePrice: Money;
  // Optional: trim-level base prices (if model has trims)
  trimBasePrices?: Record<string, Money>;

  // Packages can auto-include options and/or add their own price.
  packages?: PackageDef[];

  // Rule-driven conditional line items (rigging surcharges, etc.)
  lineItemRules?: PricingRuleDef[];

  // Notes for PDFs/build sheet
  notes?: string;
};

rendering: {
    views: ViewDef[];
    layerGroups?: LayerGroupDef[];
    galleries?: GalleryDef[];
  };

  validation?: {
    // Optional: global constraints (e.g., max total selections in a group)
    constraints?: ConstraintDef[];
  };
}

export interface StepDef {
  id: string;
  title: string;
  description?: string;
  order: number;
  sections: SectionDef[];
}

export interface SectionDef {
  id: string;
  title: string;
  description?: string;
  order: number;
  fields: FieldDef[];
}

export interface FieldDef {
  id: string;
  type: FieldType;
  label: string;
  helpText?: string;
  order: number;

  // UI behavior
  requiredByDefault?: boolean; // rules can override
  defaultValue?: BuilderValue;
  visibleByDefault?: boolean; // rules can override

  // Value constraints
  min?: number;
  max?: number;
  step?: number;
  // Options for select-like fields
  options?: OptionDef[];

  // Optional: grouping / analytics
  tags?: string[];
}

export interface OptionDef {
  id: string;
  label: string;
  code?: string; // build code/SKU

  // Pricing delta when selected (may be 0)
  price?: Money;

  // Optional: display helpers
  swatchHex?: string;
  thumbnailUrl?: string;

  // Optional: preview behavior (does not necessarily affect layered rendering)
  preview?: {
    mode: 'none' | 'gallery' | 'tooltip_image';
    galleryId?: string;
    tooltipImageUrl?: string;
  };

  // Optional: compatibility tags (can drive mappings)
  tags?: string[];
}

export type BuilderValue = string | number | boolean | null | string[];

export interface Money {
  amount: number; // store as decimal in DB; number in JS at runtime
  currency: string; // 'USD'
}

// ----------------------
// Rules DSL (compiled)
// ----------------------

export interface RuleDef {
  id: string;
  description?: string;
  priority: number; // higher wins
  when: ConditionExpr;
  then: ActionDef[];
  else?: ActionDef[];
  origin?: {
    scope: 'client' | 'model' | 'version' | 'system';
    sourceId?: string; // id of the entity that generated this rule
  };
}

export type ConditionExpr =
  | { all: ConditionExpr[] }
  | { any: ConditionExpr[] }
  | { not: ConditionExpr }
  | { eq: [fieldId: string, value: BuilderValue] }
  | { neq: [fieldId: string, value: BuilderValue] }
  | { in: [fieldId: string, values: BuilderValue[]] }
  | { contains: [fieldId: string, value: BuilderValue] }
  | { gt: [fieldId: string, value: number] }
  | { lt: [fieldId: string, value: number] };

export type ActionDef =
  | { action: 'showStep'; stepId: string }
  | { action: 'hideStep'; stepId: string }
  | { action: 'showSection'; sectionId: string }
  | { action: 'hideSection'; sectionId: string }
  | { action: 'showField'; fieldId: string }
  | { action: 'hideField'; fieldId: string }
  | { action: 'enableOption'; fieldId: string; optionId: string }
  | { action: 'disableOption'; fieldId: string; optionId: string }
  | { action: 'requireField'; fieldId: string }
  | { action: 'unrequireField'; fieldId: string }
  | { action: 'setValue'; fieldId: string; value: BuilderValue }
  | { action: 'clearValue'; fieldId: string }
  | { action: 'addLineItem'; lineItem: LineItemDef }
  | { action: 'removeLineItem'; lineItemId: string };

export interface DerivedFieldDef {
  fieldId: string;
  // Minimal v0: map-based derivation (can expand later)
  derivesFromFieldId: string;
  mapping?: Record<string, BuilderValue>; // input optionId -> output value
}
```

```
export interface PackageDef {
  id: string;
  label: string;
  code?: string;
  price?: Money; // package price add-on
  includes?: Array<{ fieldId: string; optionId: string }>;
}

export interface PricingRuleDef {
  id: string;
  priority: number;
  when: ConditionExpr;
  addLineItems: LineItemDef[];
}

export interface LineItemDef {
  id: string;
  label: string;
  code?: string;
  amount: Money;
  quantity?: number; // default 1
  category?: string; // e.g., 'rigging', 'engine', 'electronics'
}

export interface ViewDef {
  id: ViewId;
  label: string;
  width: number;
  height: number;
  layers: LayerDef[];
}

export interface LayerDef {
  id: string;
  viewId: ViewId;
  groupId?: string; // optional for admin grouping
  z: number;
  assetUrl: string; // PNG/WebP with transparency
  when: ConditionExpr;
}

export interface LayerGroupDef {
  id: string;
  label: string;
  viewId?: ViewId; // optional: group scoped to a view
}

export interface GalleryDef {
  id: string;
  label: string;
  media: Array<{ url: string; caption?: string }>;
}

export interface ConstraintDef {
  id: string;
  description?: string;
  // v0 placeholder. Implement as needed.
}
```

Appendix D: Rules DSL JSON Schema (minimal v0)

This schema is intended for validating rules authored in the admin dashboard (Payload). It matches the ConditionExpr and ActionDef shapes used by the Engine. In v0 we validate structure (not cross-references); referential integrity (fieldId/optionId exists) is enforced by compile-time validation during ModelVersion publish.

D.1 JSON Schema

```
{
  "$schema": "https://json-schema.org/draft/2020-12/schema",
  "$id": "https://launchpadcreativegroup.com/schemas/boat-builder/rules-v0.schema.json",
  "title": "Boat Builder Rules DSL (v0)",
  "type": "object",
  "additionalProperties": false,
  "properties": {
    "rules": {
      "type": "array",
      "items": { "$ref": "#/$defs/Rule" }
    }
  },
  "required": ["rules"],
  "$defs": {
    "BuilderValue": {
      "oneOf": [
        { "type": "string" },
        { "type": "number" },
        { "type": "boolean" },
        { "type": "null" },
        {
          "type": "array",
          "items": { "type": "string" }
        }
      ]
    },

    "Condition": {
      "oneOf": [
        {
          "type": "object",
          "additionalProperties": false,
          "properties": {
            "all": {
              "type": "array",
              "items": { "$ref": "#/$defs/Condition" },
              "minItems": 1
            }
          },
          "required": ["all"]
        },
        {
          "type": "object",
          "additionalProperties": false,
          "properties": {
            "any": {
              "type": "array",
              "items": { "$ref": "#/$defs/Condition" },
              "minItems": 1
            }
          },
          "required": ["any"]
        },
        {
          "type": "object",
          "additionalProperties": false,
          "properties": {
            "not": { "$ref": "#/$defs/Condition" }
          },
          "required": ["not"]
        },

        {
          "type": "object",
          "additionalProperties": false,
          "properties": {
            "eq": {
              "type": "array",
              "prefixItems": [
                { "type": "string" },
                { "$ref": "#/$defs/BuilderValue" }
              ],
              "items": false,
              "minItems": 2,
              "maxItems": 2
            }
          },
          "required": ["eq"]
        },
        {
            "type": "object",
            "additionalProperties": false,
            "properties": {
              "neq": {
                "type": "array",
                "prefixItems": [
                  { "type": "string" },
                  { "$ref": "#/$defs/BuilderValue" }
                ],
                "items": false,
                "minItems": 2,
                "maxItems": 2
              }
            },
            "required": ["neq"]
          },
          {
            "type": "object",
            "additionalProperties": false,
            "properties": {
              "in": {
                "type": "array",
                "prefixItems": [
                  { "type": "string" },
                  {
                    "type": "array",
                    "items": { "$ref": "#/$defs/BuilderValue" },
                    "minItems": 1
                  }
                ],
                "items": false,
                "minItems": 2,
                "maxItems": 2
              }
            },
            "required": ["in"]
          },
          {
            "type": "object",
            "additionalProperties": false,
            "properties": {
              "contains": {
                "type": "array",
                "prefixItems": [
              { "type": "string" },
              { "$ref": "#/$defs/BuilderValue" }
            ],
            "items": false,
            "minItems": 2,
            "maxItems": 2
          }
        },
        "required": ["contains"]
},
{
  "type": "object",
  "additionalProperties": false,
  "properties": {
    "gt": {
      "type": "array",
      "prefixItems": [
        { "type": "string" },
        { "type": "number" }
      ],
      "items": false,
      "minItems": 2,
      "maxItems": 2
    }
  },
  "required": ["gt"]
},
{
  "type": "object",
  "additionalProperties": false,
  "properties": {
    "lt": {
      "type": "array",
      "prefixItems": [
        { "type": "string" },
        { "type": "number" }
      ],
      "items": false,
      "minItems": 2,
      "maxItems": 2
    }
  },
  "required": ["lt"]
}
```

```
  ]
},

"LineItem": {
  "type": "object",
  "additionalProperties": false,
  "properties": {
    "id": { "type": "string" },
    "label": { "type": "string" },
    "code": { "type": "string" },
    "amount": {
      "type": "object",
      "additionalProperties": false,
      "properties": {
        "amount": { "type": "number" },
        "currency": { "type": "string" }
      },
      "required": ["amount", "currency"]
    },
    "quantity": { "type": "number" },
    "category": { "type": "string" }
  },
  "required": ["id", "label", "amount"]
},

"Action": {
  "oneOf": [
    {
      "type": "object",
      "additionalProperties": false,
      "properties": { "action": { "const": "showStep" }, "stepId": { "type": "string" } },
      "required": ["action", "stepId"]
    },
    {
      "type": "object",
      "additionalProperties": false,
      "properties": { "action": { "const": "hideStep" }, "stepId": { "type": "string" } },
      "required": ["action", "stepId"]
    },
    {
      "type": "object",
      "additionalProperties": false,
      "properties": { "action": { "const": "showSection" }, "sectionId": { "type": "string" } },
      "required": ["action", "sectionId"]
    },
    {
      "type": "object",
      "additionalProperties": false,
      "properties": { "action": { "const": "hideSection" }, "sectionId": { "type": "string" } },
      "required": ["action", "sectionId"]
    },
    {
      "type": "object",
      "additionalProperties": false,
      "properties": { "action": { "const": "showField" }, "fieldId": { "type": "string" } },
      "required": ["action", "fieldId"]
    },
    {
      "type": "object",
      "additionalProperties": false,
      "properties": { "action": { "const": "hideField" }, "fieldId": { "type": "string" } },
      "required": ["action", "fieldId"]
    },

    {
      "type": "object",
      "additionalProperties": false,
      "properties": {
        "action": { "const": "enableOption" },
        "fieldId": { "type": "string" },
        "optionId": { "type": "string" }
      },
      "required": ["action", "fieldId", "optionId"]
    },
    {
      "type": "object",
      "additionalProperties": false,
      "properties": {
        "action": { "const": "disableOption" },
        "fieldId": { "type": "string" },
        "optionId": { "type": "string" }
      },
      "required": ["action", "fieldId", "optionId"]
    },

    {
      "type": "object",
      "additionalProperties": false,
      "properties": { "action": { "const": "requireField" }, "fieldId": { "type": "string" } },
      "required": ["action", "fieldId"]
    },
    {
      "type": "object",
      "additionalProperties": false,
      "properties": { "action": { "const": "unrequireField" }, "fieldId": { "type": "string" } },
      "required": ["action", "fieldId"]
    },

    {
      "type": "object",
      "additionalProperties": false,
      "properties": {
        "action": { "const": "setValue" },
        "fieldId": { "type": "string" },
        "value": { "$ref": "#/$defs/BuilderValue" }
      },
      "required": ["action", "fieldId", "value"]
    },
    {
      "type": "object",
      "additionalProperties": false,
      "properties": { "action": { "const": "clearValue" }, "fieldId": { "type": "string" } },
      "required": ["action", "fieldId"]
    },

    {
      "type": "object",
      "additionalProperties": false,
      "properties": { "action": { "const": "addLineItem" }, "lineItem": { "$ref": "#/$defs/LineItem" } },
      "required": ["action", "lineItem"]
    },
    {
      "type": "object",
      "additionalProperties": false,
      "properties": { "action": { "const": "removeLineItem" }, "lineItemId": { "type": "string" } },
      "required": ["action", "lineItemId"]
    }
  ]
},

"Rule": {
  "type": "object",
      "additionalProperties": false,
      "properties": {
        "id": { "type": "string" },
        "description": { "type": "string" },
        "priority": { "type": "number" },
        "when": { "$ref": "#/$defs/Condition" },
        "then": {
          "type": "array",
          "items": { "$ref": "#/$defs/Action" },
          "minItems": 1
        },
        "else": {
          "type": "array",
          "items": { "$ref": "#/$defs/Action" }
        },
        "origin": {
          "type": "object",
          "additionalProperties": false,
          "properties": {
            "scope": { "enum": ["client", "model", "version", "system"] },
            "sourceId": { "type": "string" }
          },
          "required": ["scope"]
        }
      },
      "required": ["id", "priority", "when", "then"]
    }
  }
}
```

D.2 Notes

- This schema validates structure only. During publish/compile, run referential checks (fieldId/optionId/stepId exist, option belongs to field, etc.).
- Normalization requirements (engine): treat missing fields as null; sort arrays for multi_select; ensure consistent ids/codes across versions.
- For complex scenarios later (ranges, regex, math, computed aggregates), extend ConditionExpr with new operators under a new schema version.

Appendix E: Compile step and publish pipeline specification

E.1 Purpose and outcomes

The Compile Step converts editable CMS content (Payload collections) into an immutable, runtime-optimized ModelConfig artifact that the Builder UI consumes. Publishing a model version must always produce a deterministic compiled output so that submissions, pricing, and PDFs remain consistent over time.

The compile/publish pipeline must ensure:

- Determinism: identical inputs produce identical compiled outputs (stable ordering, stable IDs).
- Safety: invalid logic, broken references, or missing assets cannot be published.
- Performance: the Builder UI loads a single compiled config payload (plus assets) rather than many relational queries.
- Historical accuracy: published versions are immutable; submissions reference a specific published version.

E.2 Source inputs (Payload) and minimum data requirements

During editing, configuration data is stored in Payload collections (or equivalent) that support an ACF-like authoring experience. At publish time, the compiler reads the draft ModelVersion and all referenced entities.

Minimum required entities for compile:

- Client: branding, disclaimers, contact settings, default pricing/display settings.
- Model: name, modelCode, available views (profile/overhead), base price definition.
- ModelVersion (draft): steps/sections/fields/options, rule sets, layer definitions, pricing definitions, and any compatibility mappings.
- Assets: media records for any layers or galleries referenced by the version.

Best practice: keep authoring data relational for admin UX, but always compile to a single runtime payload (ModelConfig) for the builder.

E.3 Compile outputs (artifacts stored and served)

Publishing produces one or more artifacts that are written to the database (and optionally object storage) and then served to the Builder UI.

Artifacts to persist for each published ModelVersion:

- Compiled ModelConfig JSON (the single payload the builder loads).
- Asset manifest (all layer/gallery assets referenced, by view and purpose).
- Validation report (pass/fail, warnings, and a summary of checks performed).
- Deterministic hash/signature (hash of compiled config + asset version IDs) used for caching and render keys.

Important: submissions must store a snapshot of state + computed line items, and reference the published ModelVersion ID. The compiled config itself should also be retrievable later for re-rendering PDFs or debugging.

E.4 Validation stages (what must be checked before publish)

Publishing is a gated operation. Validation should run in stages so errors are actionable:

- Structural validation: schema checks for steps/fields/options and JSON Schema validation for rules (Appendix D).
- Referential integrity: all IDs referenced by rules, layers, packages, and mappings exist and are valid (e.g., optionId belongs to fieldId).
- Uniqueness and stability: field keys/ids are unique within the version; option codes are unique where required; ordering is stable.
- Rule safety checks: unknown actions/operators rejected; illegal targets rejected; max-iteration stability loop cannot oscillate for simple test states (smoke test).
- Pricing checks: all prices numeric; currency formatting rules; packages include valid items; no duplicate line-item keys.
- Rendering checks: all referenced assets exist and are reachable; z-index/order is valid; no conflicting layers within a mutually exclusive group unless intentional.
- Derived/mapping checks: mapping keys/values reference valid option IDs; default fallbacks exist where required.

Validation output should include line-level pointers (step/field/option IDs) and human-readable messages so the admin can correct issues quickly.

E.5 Normalization rules (how draft data becomes runtime data)

Normalization ensures determinism and simplifies runtime evaluation. The compiler should apply the following transformations:

- Stable identifiers: runtime keys always use stable IDs (UUID/slug). Never evaluate logic by label text.
- Stable ordering: steps, sections, fields, options, rules, and layers are sorted deterministically (explicit order fields, then stable tiebreaker).
- Index maps: build O(1) lookup maps (fieldsById, optionsById, stepsById) and precomputed relationships (optionsByFieldId).
- Default state: compute an initial state object using defaults, required base selections, and any always-on packages (if applicable).
- Compatibility tables: compile compatibility mappings into direct lookup structures (e.g., allowedOptions[fieldId][driverOptionId] = set(...)).
- Layer selection: precompile layer predicates into a normalized expression format and group layers by view (profile/overhead/detail).
- Pricing definitions: normalize price items into a line-item rule format (base + deltas + conditional surcharges) with unique keys.

E.6 Publish workflow and lifecycle states

Model versions should follow an explicit lifecycle. The simplest reliable approach is: Draft -> Published -> Archived.

- Draft: editable in admin. Rules/assets/pricing can change. Not used by public builder.
- Published: immutable. Has a compiled artifact and a hash. Used by public builder and referenced by submissions.
- Archived: immutable but no longer selectable for new builds (kept for history and reprints).

Best practice: never edit a Published version. To change pricing/logic/assets, clone it into a new Draft version, update, validate, and publish.

E.7 Runtime loading, caching, and performance

The Builder UI should load one compiled config payload per session (per model version). Use HTTP caching (ETag/If-None-Match) keyed by the compiled hash.

Recommended caching strategy:

- Config caching: serve compiled ModelConfig via an API endpoint with ETag and long max-age for immutable published versions.
- Asset caching: host images on a CDN/object storage with immutable URLs (include file hash in filename where possible).
- Render caching: server compositing uses a renderKey derived from (modelVersionId + view + normalizedState + compiledHash). Store composed outputs keyed by renderKey.

Hybrid imagery note: do not call the render endpoint on every click. Only compose on submit or on explicit 'Generate PDF / Final Render' actions.

E.8 Error handling and admin UX (so authors can fix issues)

Publishing should fail fast with a clear report. The admin UI should surface:

- Blocking errors (must fix to publish) vs warnings (publish allowed but highlighted).
- Direct links to the failing entity (step/field/option/layer) where possible.
- A minimal 'simulate state' tool that can run rules against a saved sample state and show which rules fired (v1 feature, but keep compiler hooks ready).

E.9 Suggested repo modules and function signatures

Keep compilation and runtime evaluation in shared TypeScript packages so frontend and backend stay aligned.

Recommended module layout:

- packages/engine/src/model-config.ts (ModelConfig contract, v0)
- packages/engine/src/rules/* (rule eval, actions, condition eval)
- packages/engine/src/pricing/* (line-item engine)

- packages/engine/src/render/* (layer selection, renderKey helpers)
- packages/engine/src/compiler/* (draft -> compiled transformations + validations)

Core compiler function (conceptual):

```
compileModelVersion({
  client,
  model,
  modelVersionDraft,
  assets,
}): {
  compiled: ModelConfig,
  manifest: AssetManifest,
  hash: string,
  report: ValidationReport
}
```

E.10 Example publish sequence

A publish operation should perform these steps in order:

- Load draft ModelVersion and referenced entities (steps/fields/options, rules, layers, pricing, mappings, assets).
- Run structural validation (schema) and fail on errors.
- Run referential integrity checks (IDs, option ownership, layer assets) and fail on errors.
- Normalize and compile into runtime ModelConfig (indexes, ordering, defaults, compiled predicates).
- Compute compiled hash/signature and build asset manifest.
- Persist compiled artifacts and mark version as Published (lock editing).
- Invalidate any relevant caches (config endpoint, admin previews).